Lightning Fast and Space Efficient Inequality Joins

Zuhair Khayyat[◇*] William Lucia[§] Meghna Singh[§] Mourad Ouzzani[§]
Paolo Papotti[§] Jorge-Arnulfo Quiané-Ruiz[§] Nan Tang[§] Panos Kalnis[◇]
[§]Qatar Computing Research Institute
[◇]King Abdullah University of Science and Technology (KAUST)
{zuhair.khayyat,panos.kalnis}@kaust.edu.sa, williamlucia.wl@gmail.com
{mesingh,mouzzani,ppapotti,jquianeruiz,ntang}@qf.org.qa

ABSTRACT

Inequality joins, which join relational tables on inequality conditions, are used in various applications. While there have been a wide range of optimization methods for joins in database systems, from algorithms such as sort-merge join and band join, to various indices such as B^+-tree, R^*-tree and Bitmap, inequality joins have received little attention and queries containing such joins are usually very slow. In this paper, we introduce fast inequality join algorithms. We put columns to be joined in sorted arrays and we use permutation arrays to encode positions of tuples in one sorted array *w.r.t.* the other sorted array. In contrast to sort-merge join, we use space efficient bit-arrays that enable optimizations, such as Bloom filter indices, for fast computation of the join results. We have implemented a centralized version of these algorithms on top of PostgreSQL, and a distributed version on top of Spark SQL. We have compared against well known optimization techniques for inequality joins and show that our solution is more scalable and several orders of magnitude faster.

1. ONCE UPON A TIME . . .

Bob[1], a data analyst working for an international provider of cloud services, wanted to analyze revenue and utilization trends from different regions. In particular, he wanted to find out all those transactions from the West-Coast that last longer and produce smaller revenues than any transaction in the East-Coast. In other words, he was looking for any customer from the West-Coast who rented a virtual machine for more hours than any customer from the East-Coast, but who paid less. Figure 1 illustrates a data instance for both tables. He wrote the following join query for such a task:

Q_t : SELECT east.id, west.t_id
FROM east, west
WHERE east.dur < west.time AND east.rev > west.cost;

east

	id	dur	rev	cores
r_1	100	140	12	2
r_2	101	100	12	8
r_3	102	90	5	4

west

	t_id	time	cost	cores
s_1	404	100	6	4
s_2	498	140	11	2
s_3	676	80	10	1
s_4	742	90	5	4

Figure 1: East-Coast and West-Coast transactions

Bob first ran Q_t over 200K transactions on the distributed system storing the data (System-X). Given that the input dataset is ∼1GB, he expected to have his answer in a minute or so. However, he waited for more than three hours without seeing any result. He immediately thought that this problem comes from System-X and killed the query. He then used an open-source DBMS-X to run his query. Although join is by far the most important and most studied operator in the relational algebra [**?**], Bob had to wait for over two hours until DBMS-X returned the results. He found that Q_t is processed by DBMS-X as a *Cartesian product* followed by a *selection* predicate, which is problematic due to the huge number of unnecessary intermediate results.

In the meantime, Bob heard that a big DBMS vendor was in town to highlight the power of their recently released distributed DBMS to process big data (DBMS-Y). So he visited them with a small (few KBs) dataset sample of the tables to run Q_t. Surprisingly, DBMS-Y could not run Q_t for even that small sample! He spent 45 minutes waiting while one of the DBMS-Y experts was trying to solve the issue. Bob left the query running and the vendor never contacted him again. In fact, DBMS-Y is using underneath the same open-source DBMS-X that Bob tried before. He thus understood that a simple distribution of the process does not solve his problem. Afterwards, Bob decided to call one of his friends working for a very famous DBMS vendor. His friend kindly accepted to try Q_t on their DBMS-Z, which is well reputed to deal with terabytes of data. A couple of days later, his friend came back to him with several possible ways (physical plans) to run Q_t on DBMS-Z. Nonetheless, all these query plans still had the quadratic complexity of a Cartesian product with its inherent inefficiency.

Despite the prevalence of this kind of queries in applications, such as temporal and spatial databases, and data cleaning, no off-the-shelf efficient solutions exist. There have been countless techniques to optimize the different flavors of joins in various settings [**?**]. In the general case of a theta join, one may assume that one of the relations is small enough to fit in memory; a nested loop join with the small relation stored in memory would deliver acceptable performance. However, such assumption may not hold when joining two big relations. Queries could also contain selection predicates or an equi-join with a high selectivity, which

[*]Work partially done while doing an internship at QCRI.
[1]We motivate the problem with a real-life story. Names and queries have been changed to protect confidentiality.

Proceedings of the VLDB Endowment, Vol. 8, No. 13

would reduce the size of the relations to be fed to the inequality join. However, this is not necessarily true with low-selectivity predicates, such as gender or region, where the obtained relations are still very large. Furthermore, similar to Q_t, there is a large spectrum of applications where the above two assumptions do not necessarily hold. Such applications require to join large relations using inequalities only, such as in temporal and spatial databases, and data cleaning applications. For example, in *data analysis* in a temporal database, one may want to find all employees and managers that overlapped while working in a certain company [**?**]. In data cleaning, when *detecting violations* based on denial constraints, one may want to find all pairs of tuples such that one individual (represented in the tuple) pays more taxes but earns less than another individual [**?**].

Bob then started looking at alternatives. Common ways of optimizing such queries include sort-merge joins [**?**] and interval-based indexing [**?**, **?**, **?**]. Sort merge join reduces the search space by sorting the data based on the joining attributes and merging them. However, it still has a quadratic complexity for queries with inequality join conditions only. Interval-based indexing reduces the search space of such queries even further by using bitmap interval indexing [**?**]. However, such indices require large memory space [**?**] and long index building time. Moreover, Bob would have to create multiple indices to cover all those attributes referenced in his query workload. One may build such indices at query time, but their long construction time renders such an approach impractical.

With no hope in the horizon, Bob decided to talk with his friends who happen to do research in data analytics. They happily started working on this interesting problem. After several months of hard work, they came out with IEJoin, a new algorithm that utilizes bit-arrays and positional permutation arrays to achieve fast inequality joins. Given the inherent quadratic complexity of inequality joins, IEJoin follows the *RAM locality is King* principle coined by Jim Gray. The use of memory-contiguous data structures with small footprint results in orders of magnitude performance improvement over the prior art. The basic idea of our proposal is to create a sorted array for each inequality comparison and compute their intersection, which would output the join results. The prohibitive cost of the intersection operation is alleviated through the use of a permutation array to encode positions of tuples in one sorted array *w.r.t.* the other sorted array (assuming that there are only two conditions). A bit-array is then used to emit the join results.

Contributions. We claim the following contributions:

(1) We present novel, fast and space efficient inequality join algorithms (Sections 2 and 3).

(2) We discuss two optimization techniques to significantly speed up the computation (Section 4). Specifically, we exploit Bloom filters to reduce the search space, and reorganize data to improve data locality.

(3) We describe how to implement the proposed algorithms in distributed data processing systems, such as Spark SQL [**?**] to handle very large datasets (Section 5). In particular, we use attribute metadata (*e.g.,* min and max values) to greatly reduce data shuffling.

(4) We implemented our algorithms on both PostgreSQL and Spark SQL (Section 6). We conducted an extensive experimental study by comparing against well known optimization techniques. The results show that our proposed solution is more general, scalable, and orders of magnitude faster than known prior art (Section 7).

We discuss the related work in Section 8 and conclude the paper in Section 9.

2. OVERVIEW

In this section we restrict our discussions to queries with inequality predicates only. Each predicate is of the form: A_i op B_i. Here, A_i (resp., B_i) is an attribute in relation R (resp., S), and op is an inequality operator in $\{<, >, \leq, \geq\}$. In the following, we motivate our work and give the intuition of how our algorithms work by using self-join queries.

Example 1: (Single predicate) Consider the west table in Figure 1 and an inequality self-join query Q_s as follows:

Q_s : SELECT s_1.t_id, s_2.t_id
FROM west s_1, west s_2
WHERE s_1.time $>$ s_2.time;

Query Q_s returns a set of pairs $\{(s_i, s_j)\}$ where s_i takes more time than s_j; the result is $\{(s_2, s_1), (s_2, s_3), (s_2, s_4), (s_1, s_3), (s_1, s_4), (s_4, s_3)\}$. □

A natural idea to handle an inequality join on one attribute is to leverage a sorted array. For instance, we sort west's tuples on time in ascending order into an array L_1:$\langle s_3, s_4, s_1, s_2\rangle$. We denote by $L[i]$ the i-th element in array L, and $L[i, j]$ its sub-array from position i to position j. Given a tuple s, any tuple at $L_1[k]$ ($k \in [1, i-1]$) has a time value that is less than $L_1[i]$, the position of s in L_1. Consider Example 1, tuple s_1 in position $L_1[3]$ joins with tuples in positions $L_1[1, 2]$, namely s_3 and s_4.

Example 2: (Two predicates) Let us now consider a more challenging case of a self-join with two inequality conditions:

Q_p : SELECT s_1.t_id, s_2.t_id
FROM west s_1, west s_2
WHERE s_1.time $>$ s_2.time AND s_1.cost $<$ s_2.cost;

Q_p returns pairs (s_i, s_j) where s_i takes more time but pays less than s_j; the result is $\{(s_1, s_3), (s_4, s_3)\}$. □

Similar to attribute time in Example 1, one can additionally sort attribute cost in ascending order into an array L_2:$\langle s_4, s_1, s_3, s_2\rangle$. Thus, given a tuple s, any tuple $L_2[l]$ ($l \in [j+1, n]$), where n is the size of the input relation, has higher cost than the one in s, where j is the position of s in L_2. Our observation here is as follows. For any tuple s', to form a join result (s, s') with tuple s, the following two conditions must be satisfied: (i) s' is on the left of s in L_1, *i.e., s* has a larger value for time than s', and (ii) s' is on the right of s in L_2, *i.e., s* has a smaller value for cost than s'. Thus, all tuples in the intersection of $L_1[1, i-1]$ and $L_2[j+1, n]$ satisfy these two conditions and belong to the join result. For example, s_4's position in L_1 (resp. L_2) is 2 (resp. 1). Hence, $L_1[1, 2-1] = \langle s_3\rangle$ and $L_2[1+1, 4] = \langle s_1, s_3, s_2\rangle$, and their intersection is $\{s_3\}$, producing (s_4, s_3). To get the final result, we simply need to repeat the above process for each tuple.

The challenge is how to perform the aforementioned intersection operation in an efficient manner. There already exist several indices, such as R-tree and B^+-tree, that can possibly help. R-tree is ideal for supporting two or higher dimensional range queries. However, the main shortcoming

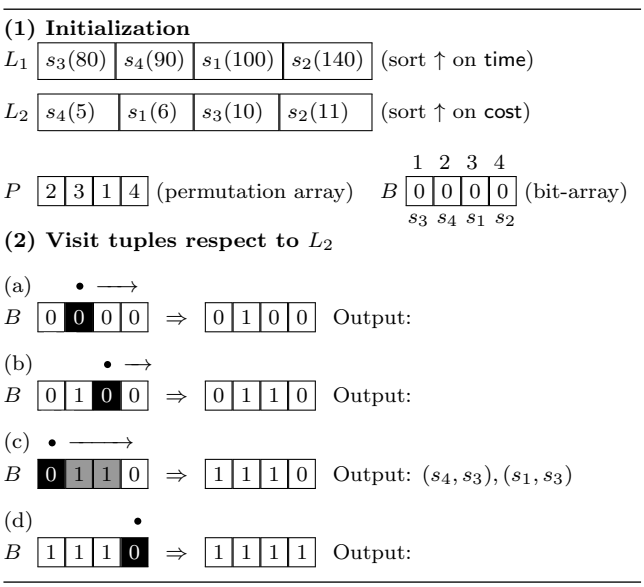


Figure 2: IEJoin process for query Q_p

of using R-trees for inequality joins is that it is unclustered; we cannot avoid random I/O access when retrieving join results. B^+-tree is a clustered index. The bright side is that for each tuple, only sequential disk scan is required to retrieve relevant tuples. However, the dark side is that we need to repeat this n times, where n is the number of tuples, which is prohibitively expensive. When confronted with such problems, one common practice is to use space-efficient and CPU-friendly indices; in this paper, we employ a bit-array.

In a nutshell, our method, namely IEJoin, sorts relation west on time and cost, creates a permutation array for cost *w.r.t.* time, and leverages a bit-array to emit join results. We will briefly present the algorithm below, and defer a detailed discussion to Section 3. Figure 2 depicts the process.

(1) *Initialization.* Sort both time and cost values in ascending order, as depicted by L_1 and L_2, respectively. While sorting, compute a permutation (reordering) array of elements of L_2 in L_1, as shown by P. For example, the first element of L_2 (*i.e.,* s_4) corresponds to position 2 in L_1. Hence, $P[1] = 2$. Initialize a bit-array B with length n and set all bits to 0, as shown by B with array indices above the rectangles and corresponding tuples below the rectangles.

(2) *Visit tuples in the order of L_2.* Scan the permutation array P and operate on the bit-array as shown below.

(a) *Visit $P[1]$.* First visit tuple s_4 (1st element in L_2) and check in P what is the position of s_4 in L_1 (*i.e.,* position 2). Then go to $B[2]$ and scan all bits in higher positions than 2. As all $B[i] = 0$ for $i > 2$, there is no tuple that satisfies the join condition of Q_p *w.r.t.* s_4. Finish this visit by setting $B[2] = 1$, which indicates that tuple s_4 has been visited.

(b) *Visit $P[2]$.* This corresponds to tuple s_1. It processes s_1 in a similar manner as s_4, without outputting any result.

(c) *Visit $P[3]$.* This visit corresponds to tuple s_3. Each non-zero bit on the right of s_3 (highlighted by grey cells) corresponds to a join result, because each marked cell corresponds to a tuple that pays less cost (*i.e.,* being visited first) but takes more time (*i.e.,* on the right side of its position). It thus outputs (s_4, s_3) and (s_1, s_3).

(d) *Visit $P[4]$.* This visit corresponds to tuple s_2. The process is similar to the above steps with an empty join result.

The final result of Q_p is the union of all the intermediate results from the above steps, *i.e.,* $\{(s_4, s_3), (s_1, s_3)\}$.

There are few observations that make our solution appealing. First, there are many efficient techniques for sorting large arrays, *e.g.,* GPUTeraSort [**?**]. In addition, after getting the permutation array, we only need to sequentially scan it once. Hence, we can store the permutation array on disk, instead of memory. Only the bit-array is required to stay in memory, to avoid random disk I/Os. Thus, to execute queries Q_s and Q_p on 1 billion tuples, we only need 1 billion bits (*i.e.,* 125 MB) of memory space.

3. CENTRALIZED ALGORITHMS

We now detail our novel inequality join algorithms based on sorting, permutation arrays, and bit-arrays. We will first discuss the case with two relations and only with operators in $\{<, >, \leq, \geq\}$, and then describe its extension to support multiple join conditions in Section 3.1. The special case of self-joins is presented in Section 3.2.

3.1 IEJoin

Algorithm. IEJoin, is shown in Algorithm 1. It takes a query Q with two inequality join conditions as input and returns a set of result pairs. It first sorts the attribute values to be joined (lines 3-6), computes the permutation array (lines 7-8) and two offset arrays (lines 9-10). Each element of an offset records the relative position from L_1 (resp. L_2) in L_1' (resp. L_2'). The algorithm also sets up the bit-array (line 11) as well as the result set (line 12). In addition, it sets an offset variable to distinguish between the inequality operators with or without equality conditions (lines 13-14). It then visits the values in L_2 in the desired order, which is to sequentially scan the permutation array from left to right (lines 15-22). For each tuple visited in L_2, it first sets all bits for those t in T' whose Y' values are smaller than the Y value of the current tuple in T (lines 16-18), *i.e.,* those tuples in T' that satisfy the second join condition. It then uses the other offset array to find those tuples in T' that also satisfy the first join condition (lines 19-22). It finally returns all join results (line 23). Let us illustrate this algorithm with the following example.

Example 3: Figure 3 shows how Algorithm 1 works for Q_t (from Section 1). It first does the initialization (step (1) in the figure). For example, when visiting the first item in L_2 (r_3) in step (2)(a), it first finds its relative position in L_2' at step (2)(a)(i). Then it visits all tuples in L_2' whose cost values are no larger than r_3[rev] at step (2)(a)(ii). Afterwards, it uses the relative position of r_3[dur] at L_1' (step (2)(a)(iii)) to populate all join results (step (2)(a)(iv)). The same process sequentially applies to r_1 (step (2)(b)) and r_2 (step (2)(c)), and the only result is returned at step (2)(c)(v). □

Correctness. It is easy to check that the algorithm will terminate and that each result in join_result satisfies the join condition. For *completeness*, observe the following. For any tuple pair (r_i, s_j) that should be a result, s_j will be visited first and its corresponding bit is set to 1 (lines 17-18). Afterwards, r_i will be visited and the result (r_i, s_j) will be identified (lines 20-22) by the algorithm.

Algorithm 1: IEJoin

```
   input  : query Q with 2 join predicates t1.X op1 t2.X' and
            t1.Y op2 t2.Y', tables T, T' of sizes m and n resp.
   output: a list of tuple pairs (ti, tj)
1  let L1 (resp. L2) be the array of X (resp. Y) in T
2  let L1' (resp. L2') be the array of X' (resp. Y') in T'
3  if (op1 ∈ {>, ≤}) sort L1, L1' in descending order
4  else if (op1 ∈ {<, ≥}) sort L1, L1' in ascending order
5  if (op2 ∈ {>, ≤}) sort L2, L2' in ascending order
6  else if (op2 ∈ {<, ≥}) sort L2, L2' in descending order
7  compute the permutation array P of L2 w.r.t. L1
8  compute the permutation array P' of L2' w.r.t. L1'
9  compute the offset array O1 of L1 w.r.t. L1'
10 compute the offset array O2 of L2 w.r.t. L2'
11 initialize bit-array B' (|B'| = n), and set all bits to 0
12 initialize join_result as an empty list for tuple pairs
13 if (op1 ∈ {≤, ≥} and op2 ∈ {≤, ≥}) eqOff = 0
14 else eqOff = 1
15 for (i ← 1 to m) do
16     off2 ← O2[i]
17     for j ← O2[i − 1] to O2[i] do
18         B'[P'[j]] ← 1
19     off1 ← O1[P[i]]
20     for (k ← off1 + eqOff to n) do
21         if B'[j] = 1 then
22             add tuples w.r.t. (L2[i], L2'[k]) to join_result
23 return join_result
```

Complexity. Sorting arrays and computing their permutation array is in $O(m \cdot log\, m + n \cdot log\, n)$ time, where m and n are the sizes of the two input relations (lines 3-8). Computing the offset arrays will take linear time using sort-merge (lines 9-10). The outer loop will take $O(m \cdot n)$ time (lines 15-22). Hence, the total time complexity of the algorithm is $O(m \cdot log\, m + n \cdot log\, n + m \cdot n)$. It is straightforward to see that the total space complexity is $O(m + n)$.

Multiple join conditions. For more than two join predicates on a single inequality join, we simply pick two inequality predicates and apply IEJoin. We then filter the materialized results and evaluate the remaining predicates. This approach has very low memory footprint and it is a standard solution in relational databases. A query optimizer will have to decide which predicate to process first based on the selectivity of different predicates. In Section 6, we explain how we integrate IEJoin into existing query optimizers.

3.2 IESelfJoin

In this section, we present the algorithm for self-join queries with two inequality operators. While IEJoin can be used, IESelfJoin is more efficient for self-joins since it uses two sorted arrays instead of four.

Algorithm. IESelfJoin (Algorithm 2) takes a self-join inequality query Q as input, and returns a set of result pairs.

The algorithm first sorts the two lists of attributes to be joined (lines 2-5), computes the permutation array (line 6), and sets up the bit-array (line 7) as well as the result set (line 8). It also sets an offset variable to distinguish inequality operators with or without equality (lines 9-10). It then visits the values in L_2 in the desired order, which is to sequentially scan the permutation array from left to right (lines 11-16). For each tuple visited in L_2, it needs to find all tuples whose X values satisfy the join condition. This is performed by first locating its corresponding position in L_1 via

Algorithm 2: IESelfJoin

```
   input  : query Q with 2 join predicates t1.X op1 t2.X and
            t1.Y op2 t2.Y, table T of size n
   output: a list of tuple pairs (ti, tj)
1  let L1 (resp. L2) be the array of column X (resp. Y)
2  if (op1 ∈ {>, ≤}) sort L1 in descending order
3  else if (op1 ∈ {<, ≥}) sort L1 in ascending order
4  if (op2 ∈ {>, ≤}) sort L2 in ascending order
5  else if (op2 ∈ {<, ≥}) sort L2 in descending order
6  compute the permutation array P of L2 w.r.t. L1
7  initialize bit-array B (|B| = n), and set all bits to 0
8  initialize join_result as an empty list for tuple pairs
9  if (op1 ∈ {≤, ≥} and op2 ∈ {≤, ≥}) eqOff = 0
10 else eqOff = 1
11 for (i ← 1 to n) do
12     pos ← P[i]
13     for (j ← pos + eqOff to n) do
14         if B[j] = 1 then
15             add tuples w.r.t. (L1[j], L1[i]) to join_result
16     B[pos] ← 1
17 return join_result
```

looking up the permutation array (line 12). Since the bit-array and L_1 have a one-to-one positional correspondence, the tuples on the right of pos will satisfy the join condition on X (lines 13-15), and these tuples will also satisfy the join condition on Y if they have been visited before (line 14). Such tuples will be joined with currently visited tuple as results (line 15). Afterwards, the visited tuple will also be marked (line 16). It finally returns all join results (line 17).

Note that the different sorting orders, *i.e.,* ascending or descending for attribute X and Y in lines 2-5, are chosen to satisfy various inequality operators. One may observe that if the database contains duplicated values, when sorting one attribute X, its corresponding value in attribute Y should be considered, and vice versa, in order to preserve both orders for correct join result. Hence, in IESelfJoin, when sorting X, we use an algorithm that also takes Y as the secondary key. Specifically, when some X values are equal, their sorting orders are decided by their Y values (lines 2-3), similarly for the other way around (lines 4-5). Please refer to the example in Section 2 for query Q_p using IESelfJoin.

Correctness. It is easy to check that the algorithm will terminate and that each result in join_result satisfies the join condition. For *completeness*, observe the following. For any tuple pair (t_1, t_2) that should be in the result, t_2 is visited first and its corresponding bit is set to 1 (line 16). Afterwards, t_1 is visited and the result (t_1, t_2) is identified (lines 14-15) by IESelfJoin.

Complexity. Sorting two arrays and computing their permutation array is in $O(n \cdot log\, n)$ time (lines 2-8). Scanning the permutation array and scanning the bit-array for each visited tuple run in $O(n^2)$ time (lines 11-16). Hence, in total, the time complexity of IESelfJoin is $O(n^2)$. It is easy to see that the space complexity of IESelfJoin is $O(n)$.

4. OPTIMIZATION

We discuss two optimization techniques for our inequality join algorithms. The first one is to use indices to improve the lookup performance for the bit-array (Section 4.1). The second one is to union arrays, so as to improve data locality and reduce the data to be loaded into the cache (Section 4.2).

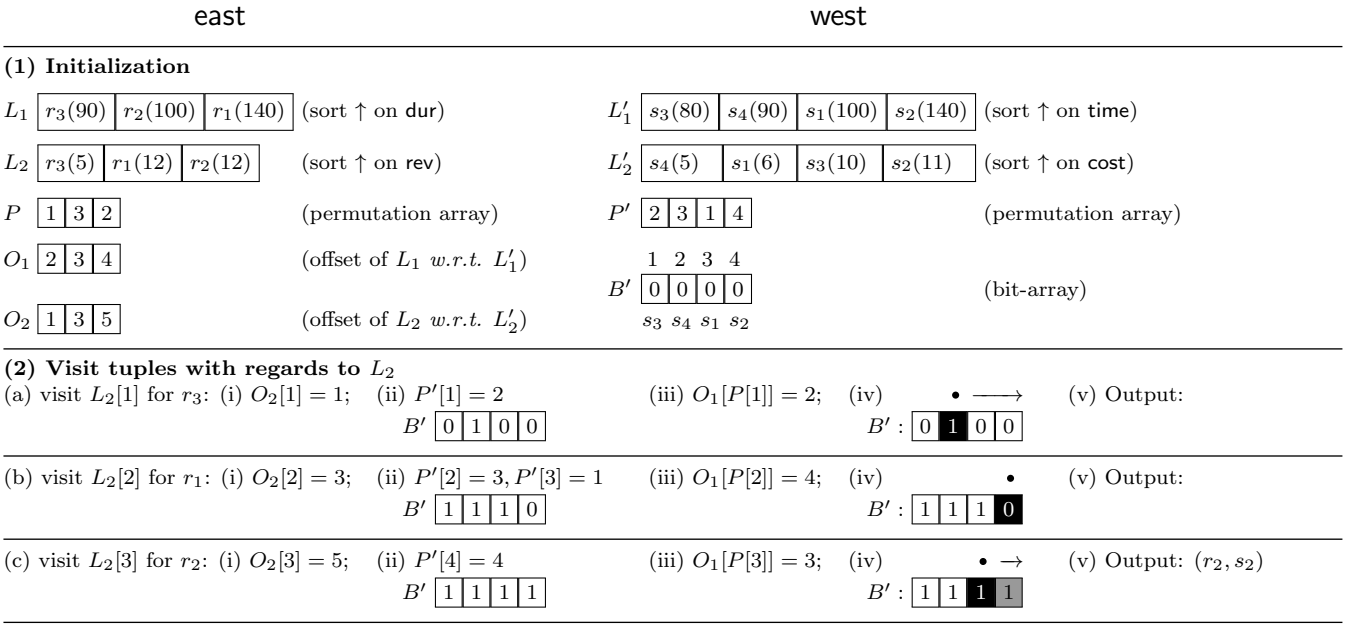


Figure 3: IEJoin process for query Q_t

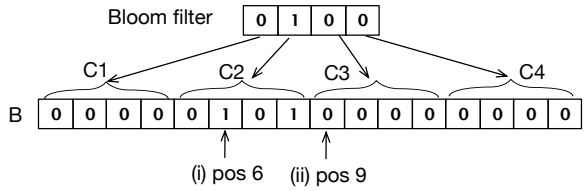


Figure 4: Example of using a Bloom filter

4.1 Bloom Filter to Improve Bit-array Scan

An analysis on both IEJoin and IESelfJoin shows that, for each value being visited (*i.e.,* lines 20-22 in Algorithm 1 and lines 13-15 in Algorithm 2), we need to scan all the bits on the right of the current position. When the query selectivity is high, this is unavoidable for producing the correct results. However, when the query selectivity is low, iteratively scanning a long sequence of 0's will be a performance bottleneck. We thus adopt a Bloom filter to guide which part of the bit-array should be visited.

Given a bit-array B with size n and a predefined chunk size c, our *Bloom filter* is a bit-array with size $\lceil n/c \rceil$ where each bit corresponds to a chunk in B, with 1 indicating that the chunk contains at least a 1 and 0 otherwise.

Example 4: Consider the bit-array B in Figure 4. Assume that the chunk size $c = 4$. The bit-array B will be partitioned into four chunks C_1–C_4. Its Bloom filter is shown above B in the figure and consists of 4 bits. We consider two cases. Case (i): visit $B[6]$, in which case we need to find all the 1's in $B[i]$ for $i > 6$. The Bloom filter tells that only chunk 2 needs to be checked, and it is safe to ignore chunks 3 and 4. Case (ii): visit $B[9]$, the Bloom filter can tell that there is no need to scan B, since there cannot be any $B[j]$ where $B[j] = 1$ and $j > 9$. □

4.2 Union Arrays on Join Attributes

In testing Algorithm 1, we found that there are many cache loads and stores. A deeper analysis of the algorithm shows that the extra cache loads and stores may be caused by cache misses when sequentially visiting different arrays. Take Figure 3 for example. In Step (2)(a), we visited arrays L_2, O_2, P', P and O_1 in sequence, with each causing at least one cache miss. Step (2)(b) and Step (2)(c) show a similar behavior. An intuitive solution is to merge the arrays on join attributes and sort them together. Again, consider Figure 3. We can merge L_1 and L'_1 into one array and sort them. Similarly, we can merge L_2 and L'_2, and P and P'. Also, O_1 and O_2 are not needed in this case, and B' needs to be extended to be aligned with the merged arrays. This solution is quite similar to IESelfJoin discussed in Section 3.2. However, we need to prune join results for tuples that come from the same table. This can be easily done using a Boolean flag for each position, where 0 (resp., 1) denotes that the corresponding value is from the first (resp. the second) table. Our experiments (Section 7.2) show that the simple union operation can significantly reduce the number of cache misses, and thus improve the total execution time.

5. DISTRIBUTED INEQUALITY JOINS

We present a distributed version of the IEJoin along the same lines of state-of-the-art general purpose distributed data processing systems, such as Hadoop's MapReduce [**?**] and Spark [**?**]. Our goal is twofold: (i) scale our algorithm to very large input relations that do not fit into the main memory of a single machine and (ii) improve efficiency even further. We assume that work scheduling and data block assignment are handled by any general purpose resource manager, such as YARN (`http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`) and Mesos (`http://mesos.apache.org`).

The simplest approach for running IEJoin in a distributed setting is to: (i) construct k data blocks of each input relation; (ii) apply Cartesian product (or self-Cartesian product for a single relation input) on the data blocks; and (iii) run IEJoin (either on a single table or two tables input) on the k^2 data block pairs. However, this would generate large amount of network traffic and some data block pairs may not

Algorithm 3: Distributed IEJoin

```
   input  : Query q, Table t_in1,Table t_in2
   output: Table t_out
1  //Pre-processing
2  DistT1 ← read t_in1 in distributed blocks
3  DistT2 ← read t_in2 in distributed blocks
4  foreach row r ∈ DistT1 and DistT2 do
5      r ← global unique ID
6  DistT1 ← sort and equally partition DistT1 rows
7  DistT2 ← sort and equally partition DistT2 rows
8  forall the block b_i ∈ DistT1 do
9      D1_i ← pivot and reference values in b_i
10     MT1_i ← min and max values in pivot and
11                 reference lists of D1_i
12 forall the block b_j ∈ DistT2 do
13     D2_j ← pivot and reference values in b_j
14     MT2_j ← min and max values in pivot and
15                 reference lists of D2_j
16 Virt ← all block combinations of MT1 and MT2
17 forall the (MT1_i, MT2_j) pairs ∈ Virt_ij do
18     if MT1_i ⋂ MT2_j then
19         MData_ij ← blocks from D1_i and D2_j
20 //IEJoin function
21 forall the block pairs (D1_i, D2_j) ∈ MData_ij do
22     RowIDResult_ij ← IEJoin(q,D1_i,D2_j)
23 //Post-processing
24 forall the rowID pairs (i, j) ∈ RowIDResult_ij do
25     row_i ← row id i in DistT1
26     row_j ← row id j in DistT2
27     t_out ← merge(row_i,row_j)
```

necessarily generate results. A naive work-around would be to reduce the number of blocks for each relation to maximize the usability of each data block. However, very large data blocks introduce work imbalance and require larger memory space for each worker.

We solve the above distribution challenges by introducing efficient *pre-processing* and *post-processing* phases. These two phases allow us to reduce communication overhead and memory footprint for each node, without modifying the data block size. The pre-processing phase generates space-efficient data blocks for the input relation(s); predicts which pair of data blocks may report query results; and copies and transfers through the network only useful pairs of data blocks. IEJoin, in the distributed version, returns the join results as a pair of rowIDs instead of returning the actual rows. It is the responsibility of the post-processing phase to materialize the final results by resolving the rowIDs into actual relation rows. We use the internal rowIDs of Spark SQL to uniquely identify different rows since the input relations may not have a unique row identifier. We summarize in Algorithm 3 the implementation of the distributed algorithm when processing two input tables. We omit the discussion about single relation input since it is straightforward to follow. The distributed join process is composed of three main phases (the pre-processing, IEJoin, and post-processing phases), which are described below.

Distributed pre-processing. After assigning unique rowIDs to each input row (lines 2-5), the pre-processing step globally sorts each relation and partitions each sorted relation to k equally-sized partitions, where the number of partitions depends on the relation size and default block size (lines 6-7). For example, if the default block size is b and the relation input size is M, the number of partitions is $\lceil \frac{M}{b} \rceil$. Note that the global sorting maximizes data locality within the partitions, which in turn increases the overall runtime efficiency. This is because global sorting partially answers one of the inequality join conditions, where it physically moves tuples closer toward their candidate pairs. In other words, global sorting increases the efficiency of block pairs that generate results, while block pairs that do not produce results can be filtered out before actually processing them. After that, for each sorted partition, we generate a single data block that stores only the attribute values referenced in the join conditions in a list. These data blocks $D1$ and $D2$ do not store the actual relation rows in order to reduce the network overhead and reduce its memory footprint. This follows the semi-join principle. We also extract metadata that contain the block ID, and the min/max values of each referenced attribute value from each data block (lines 8-15). Then, we create $k_{MT1} \times k_{MT2}$ virtual block combinations and filter out block combinations that do not generate results (lines 16-19). Notice that blocks with non-intersecting min-max values do not produce results.

Distributed IEJoin. After pre-processing, we obtain a list of overlapping block pairs. We simply run IEJoin (either for a single or two relations) for each of these pair blocks in parallel. Specifically, we merge-sort the attribute values in $D1$ and $D2$ and run IEJoin over the merged block. The permutation and bit arrays generation are similar to the centralized version. However, the distributed IEJoin does not have access to the actual relation rows. Therefore, each parallel IEJoin instance outputs a pair of rowIDs that represents the joined rows (lines 21-22).

Distributed post-processing. In the final step, we materialize the result pairs by matching each rowID-pair, from the output of the distributed IEJoin, with the rowIDs of $DistT1$ and $DistT2$ (lines 24-27). We run this post-processing phase in parallel, as a distributed hash join based on the rowIDs, to speed up the materialization of the final join results.

6. IMPLEMENTATION DETAILS

We now describe the integration of our algorithms into PostgreSQL (Section 6.1) and Spark SQL (Section 6.2).

6.1 PostgreSQL

PostgreSQL processes a query in three stages: *parsing*, *planning*, and *execution*. Parsing extracts relations and predicates and creates query parse trees. Planning creates query plans and invokes the query optimizer to select a plan with the smallest estimated cost. Execution runs the selected plan and emits the output.

Parsing and Planning. PostgreSQL uses merge and hash join operators for equijoins and naive nested loop for inequality joins. PostgreSQL looks for the most suitable join operator for each join predicate. We extend this check to verify if it is IEJoin-able by checking if a predicate contains a scalar inequality operator. If so, we save the operator's oid in the data structure associated with the predicate. For each operator and ordered pair of relations, the list of predicates that the operator can handle is created. For example, two equality predicates over the same pair of relations are associated to one hash join operator.

Sort1	idx	time	cost	pos
s_1	3	80	10	1
s_2	4	90	5	2
s_3	1	100	6	3
s_4	2	140	11	4

Sort2	idx	time	cost	pos
s_1	4	90	5	2
s_2	1	100	6	3
s_3	3	80	10	1
s_4	2	140	11	4

Figure 5: Permutation array creation for self-join Q_p

Next, the Planner estimates the execution cost for possible join plans. Every node in the plan has a base cost, which is the cost of executing the previous nodes, plus the cost for the actual node. Using existing PostgreSQL methods, we added a cost function for our operator; it is evaluated as the sum of the cost for sorting inner and outer relations, CPU cost for evaluating all output tuples (approximated based on the IEJoin-predicates), and the cost of evaluating additional predicates for each tuple (*i.e.,* the ones that are not involved in the actual join). Next, PostgreSQL selects the plan with the lowest cost.

Execution. At the executor, incoming tuples from outer and inner relations are stored into *TupleTableSlot* arrays. These copies of the tuples are required as PostgreSQL may not have the content of the tuple at the same pointer location when the tuple is sent for the final projection. This step is a platform-specific overhead that is required to produce an output. The outer relation (of size N) is parsed first, followed by the inner relation (of size M). If the inner join data is identical to the corresponding outer join data (self-join), we drop the inner join data and the data structure has size N instead of $2N$.

We illustrate in Figure 5 the data structure and the permutation array computation with an example for the self-join Q_p. The data structure is initialized with an index (idx) and a copy of the attributes of interest (time and cost for Q_p). Next, the data is sorted for the first predicate (time) using the system function *qsort* with special comparators (as defined in Algorithm 1) to handle cases where two values for a predicate are equal. The result of the first sort is reported at the left-hand side of Figure 5. The last column (pos) is now filled with the ordering of the tuples according to this sorting. As a result, we create a new array to store the index values for the first predicate. We use this array to select the tuple IDs at the time of projecting tuples. The tuples are then ordered again according to the second predicate (cost), as reported in the right-hand side of Figure 5. After the second sorting, the new values in pos are the values for the permutation array, denoted by *perm*.

Finally, we create and traverse a bit-array B of size $(N + M)$ (N in case of self-join) along with a *Bloom filter* bit-array, as discussed in Section 4.1. If the traversal finds a set bit, the corresponding tuples are sent for projection. Additional predicates (if any) are evaluated at this stage, and, if the conditions are satisfied, tuples are projected.

6.2 Spark SQL

Spark SQL [**?**] allows users to query structured data on top of Spark [**?**]. It stores the input data as a set of in-memory Resilient Distributed Datasets (RDD). Each RDD is partitioned into smaller cacheable blocks, where each block fits in the memory of a single machine. Spark SQL takes as input the datasets location(s) in HDFS and an SQL query, and outputs an RDD that contains the query result. The default join operation in Spark SQL is inner join. When passing a join query to Spark SQL, the optimizer searches for equality join predicates that can be used to evaluate the inner join operator as a hash-based physical join operator. If there are no equality join predicates, the optimizer translates the inner join physically to a Cartesian product followed by a selection predicate.

We implemented the distributed version of IEJoin as a new Spark SQL physical join operator. To make the optimizer aware of the new operator, we added a new rule to recognize inequality conditions. The rule uses the first two inequality conditions for the IEJoin operator. In case of additional inequality join conditions, it evaluates them as a post selection operation on the output of the first two join conditions. The distributed operator utilizes Spark RDD operators to process the algorithm in distributed fashion. As a result, the distributed IEJoin operator depends on Spark's memory management to store the user's input relation. If the result does not fit in the memory of a single machine, we temporarily store the result into HDFS. After all IEJoin instances finish writing into HDFS, the distributed operator passes the HDFS file pointer to Spark, which constructs a new RDD of the result and passes it to Spark SQL.

Dataset	Number of rows	Size
Employees	10K – 500M	300KB – 17GB
Employees2	1B – 6B	34GB – 215GB
Events	10K – 500M	322KB – 14GB
Events2	1B – 6B	32GB – 202GB
MDC	24M	2.4GB
Cloud	470M	28.8GB

Table 1: Size of the datasets

7. EXPERIMENTAL STUDY

In this section, we evaluate IEJoin with several datasets on a set of inequality queries (Section 7.1). We study the effect of sorting and caching (Section 7.2). We then compare IEJoin with existing systems on both a centralized (Section 7.3) and a distributed environment (Section 7.4).

7.1 Datasets, Queries, and Algorithms

Datasets. We used both synthetic and real-world data (summarized in Table 1) to evaluate our algorithms.

(1) Employees. A dataset that contains employees' salary and tax information [**?**] with eight attributes: state, married, dependents, salary, tax, and three others for notes. The relation has been populated with real-life data: tax rates, income brackets, and exemptions for each state in the USA have been manually collected to generate synthetic tax records. We used the following self-join query to identify anomalies [**?**]:

Q_1 :
```
SELECT r.id, s.id
FROM Employees r, Employees s
WHERE r.salary < s.salary AND r.tax > s.tax;
```

The above query returns a set of employee pairs, where one employee earns higher salary than the other but pays less tax. To make sure that we generate output for Q_1, we selected 10% random rows and increased their tax values. Employees2 is a group of larger input datasets with up to 6 Billion records, but with only 0.001% random changes to tax values. The higher selectivity is used to test the distributed algorithm on large input files.

(2) Events. A synthetic dataset that contains start and end time information for a set of independent events. Each event

contains the name of the event, event ID, number of attending people, and the sponsor ID. We used this dataset with a self-join query that collects pairs of overlapping events:

Q_2 : SELECT r.id, s.id
FROM Events r, Events s
WHERE r.start $\leq$ s.end AND r.end $\geq$ s.start
AND r.id $\neq$ s.id;

Again, to make sure we generate output for Q_2, we selected 10% random events and extended their end values. We also generate Events2 as larger datasets with up to 6 Billion records, but with 0.001% extended random events.

(3) MOBILE DATA CHALLENGE (MDC). This is a 50GB real dataset [**?**] that contains behavioral data of nearly 200 individuals collected by Nokia Research (`https://www.idiap.ch/dataset/mdc`). The dataset contains physical locations, social interactions, and phone logs of the participating individuals. We used two relations, Shops and Persons, from the dataset with the following join query that, for all shops, looks for all persons that are close to a shop up to a distance c along the x-axis (xloc) and the y-axis (yloc):

Q_3 : SELECT s.name, p.name
FROM Shops s, Persons p
WHERE s.xloc $-$ c $<$ p.xloc AND s.xloc $+$ c $>$ p.xloc
AND s.yloc $-$ c $<$ p.yloc AND s.yloc $+$ c $>$ p.yloc;

(4) CLOUD [**?**]. A real dataset that contains cloud reports from 1951 to 2009, through land and ship stations (`ftp://cdiac.ornl.gov/pub3/ndp026c/`). We used a self-join query Q_4, similar to Q_3, to compute for every station all stations within a distance $c = 10$. Since the runtime for Q_3 and Q_4 is dominated by the output size, we mostly used them for scalability analysis in the distributed case.

Centralized Systems. We evaluated the following centralized systems in our experiments:

(1) PG-IEJOIN. We implemented IEJOIN inside PostgreSQL v9.4, as discussed in Section 6.1. We compare it against the baseline systems below.

(2) PG-ORIGINAL. We use PostgreSQL v9.4 as a baseline since it is the most widely used open source DBMS. We ran automatic configuration tuning with *pgtune* [**?**] to maximize the benefit from large main memory.

(3) PG-BTREE & PG-GiST. For optimization purposes, we use indices for Q_1 and Q_2 with two alternative approaches: a B-tree index and GiST. For PG-BTREE, we define a B-tree index for each attribute in a query. For PG-GiST, we use the GiST access method built inside PostgreSQL, which considers arbitrary indexing schemes and automatically selects the best technique for the input relation. Although Q_1 and Q_2 appear similar, they require different data representation to be able to index them using GiST. The inequality attributes in Q_1 are independent, each condition forms a single open interval. However, the inequality attributes in Q_2 are dependent, together they form a single closed interval. To use GiST in Q_1, we had to convert salary and tax attributes into a single geometric point data type SalTax, as shown in Q_{1i}. Similarly for Q_2, we converted start and end attributes into a single range data type StartEnd, as shown in Q_{2i}.

Q_{1i} : SELECT r.id, s.id
FROM Employees r, Employees s
WHERE r.SalTax $>^{\wedge}$ s.SalTax
AND r.SalTax $\gg$ s.SalTax;

Q_{2i} : SELECT r.id, s.id
FROM Events r, Events s
WHERE r.StartEnd && s.StartEnd AND r.id $\neq$ s.id;

In the rewriting of the above queries in PG-GiST, operator "$>^{\wedge}$" corresponds to "*is above?*", operator "$\gg$" means "*is strictly right of?*", and operator "&&" indicates "*overlap?*". For geometric and range type, GiST uses a Bitmap index to optimize its data access with large datasets.

(4) MONETDB. We used MonetDB Database Server Toolkit v1.1 (Oct2014-SP2), which is an open-source column-oriented database, in a disk partition of size 669GB.

(5) DBMS-X. We used a leading commercial centralized relational database.

Single node experimental setup. For the centralized evaluation, we used a Dell Precision T7500 equipped with two 64-bit quad-core Intel Xeon X5550 (8 physical cores and 16 CPU threads) and 58GB RAM.

Distributed systems. For these experiments, we used the following systems:

(1) SPARK SQL-IEJOIN. We implemented IEJOIN inside Spark SQL v1.0.2 (`https://spark.apache.org/sql/`), as detailed in Section 6.2. We evaluated the performance of our techniques against the baseline systems below.

(2) SPARK SQL & SPARK SQL-SM. Spark SQL is the default implementation in Spark SQL. Spark SQL-SM is an optimized version based on distributed sort-merge join in [**?**]. It contains three phases: *partitioning*, *sorting*, and *joining*. Partitioning selects a join attribute to distribute the data based on some statistics, *e.g.,* cardinality estimation. Sorting sorts each partition into many sorted lists, each list corresponds to an inequality condition. Finally, we apply a distributed sort merge join over the sorted lists to produce results. We also improve the above method by pruning the non-overlapping partitions to be joined.

(3) DPG-BTREE & DPG-GiST. We use a commercial version of PostgreSQL with distributed query processing. This allows us to compare SPARK SQL-IEJOIN to a distributed version of PG-BTREE and PG-GiST.

Multi-node experimental setup. We use a compute cluster of 17 Shuttle SH55J2 machines (1 master with 16 workers) equipped with Intel i5 processors with 16GB RAM, and connected to a high-end Gigabit switch.

7.2 Parameters Setting

We show the effect of the two optimizations (Section 4), as well as the effect of global sorting (Section 5).

Bloom filter. We run query Q_2 on 10M tuples to show the performance gain of using a Bloom filter. Results are shown in Table 2. Note that the chunk size is an optimization parameter that is machine specific. For this experiment, L1 cache was 256KB. Intuitively, the larger the chunk size the better. However, a very large chunk size defeats the purpose of using Bloom filters to reduce the bit-array scanning overhead. The experiment shows that the performance gain is 3X between 256 bits and 1,024 bits and 1.5X between 1,024 bits and 4,096 bits. Larger chunk sizes show worse performance, as shown with chunk size of 16,384 bits.

Union arrays. To show the performance gain due to the union optimization, we run IEJOIN, with and without the union array, using 10M tuples from the Events dataset. We

Chunk (bit)	1	64	256	1024	4096	16384
Time (sec)	>1 day	1623	896	296	158	232

Table 2: Bloom filters on 10m rows (Events data)

Parameter (M/sec)	**IEJoin** (union)	**IEJoin**
cache-references	6.5	8.4
cache-references-misses	3.9	4.8
L1-dcache-loads	**459.9**	**1,240.6**
L1-dcache-load-misses	8.7	10.9
L1-dcache-stores	**186.8**	**567.5**
L1-dcache-store-misses	1.9	1.9
L1-dcache-prefetches	4.9	7.0
L1-dcache-prefetches-misses	2.2	2.7
LLC-loads	5.1	6.6
LLC-load-misses	2.9	3.7
LLC-stores	3.8	3.7
LLC-store-misses	1.1	1.2
LLC-prefetches	3.1	4.1
LLC-prefetch-misses	2.2	2.9
dTLB-loads	**544.4**	**1,527.2**
dTLB-load-misses	0.9	1.6
dTLB-stores	**212.7**	**592.6**
dTLB-store-misses	0.1	0.1
Total time (sec)	**125**	**325**

Table 3: Cache statistics on 10m rows (Events data)

collect the following statistics, shown in Table 3: (i) L1 data caches (dcache), (ii) last level cache (LLC), and (iii) data translation lookaside buffer (dTLB). Note that the optimized algorithm with union arrays is 2.6 times faster than the original one. The performance gain in the optimized version is due to the lower number of cache loads and stores (L1-dcache-loads, L1-dcache-stores, dTLB-loads and TLB-stores), which is 2.7 to 3 times lower than the original algorithm. This behavior is expected since the optimized IEJoin has fewer arrays compared with the original version.

Global sorting on distributed IEJoin**.** As presented in Algorithm 3, the distributed version of our algorithm applies global sorting at the pre-processing phase (lines 6-7). In this experiment, we compare the performance of Q_1 and Q_2 with and without global sorting. Figure 6 shows the results of this experiment. At a first glance, one may think that the global sorting affects the performance of distributed IEJoin as it requires shuffling data through the network. However, global sorting improves the performance of the distributed algorithm by 2.4 to 2.9 times. This is because global sorting allows us to filter out block-pair combinations that do not generate results. We also observe that the time required by the IEJoin process itself is one order of magnitude faster when using global sorting.

We further breakdown the runtime of Q_1 and Q_2 in Table 4 to measure the impact of global sorting. Here, the pre-processing time includes the data loading from HDFS, global sorting, partitioning, and block-pairs materialization. Even though global sorting increases the overhead of the pre-processing phase, we observe that the runtime for this phase is at least 30% less compared with the case without global sorting due to the reduced network overhead from eliminating unnecessary block pairs. The results confirm the above observation: IEJoin is one order of magnitude faster when pre-processing includes global sorting. This greatly reduces the network overhead and increases the memory locality in the block combinations that are passed to our algorithm.

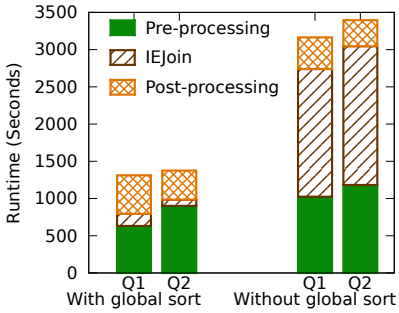


Figure 6: IEJoin **using 100m rows on 6 workers**

Query	**Pre-process**	IEJoin	**Post-process**	**Total**
With global sorting				
Q_1	632	**162**	519	1,313
Q_2	901	**84**	391	1,376
Without global sorting				
Q_1	1,025	1,714	426	3,165
Q_2	1,182	1,864	349	3,395

Table 4: Time breakdown (secs) of Figure 6

Based on the above experiments, in the following tests we used 1,024 bits as the default chunk size, union arrays, and global sorting for distributed IEJoin.

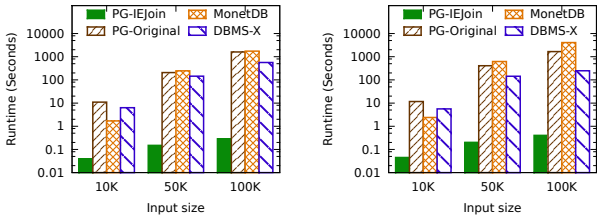
7.3 Single-node Experiments

In this set of experiments, we study the efficiency of IEJoin on datasets that fit the main memory of a single compute node and compare its performance with alternative centralized systems.

IEJoin **vs. baseline systems.** Figure 7 shows the results for queries Q_1 and Q_2 in a centralized environment, where the x-axis represents the input size in terms of the number of tuples, and the y-axis represents the corresponding running time in seconds. The figure reports that PG-IEJoin outperforms all baseline systems by more than one order of magnitude for both queries and for every reported dataset input size. In particular, PG-IEJoin is up to more than three (resp., two) orders of magnitude faster than PG-Original and MonetDB (resp., DBMS-X). We can clearly see that the baseline systems cannot compete with PG-IEJoin since they all use the classic Cartesian product followed by a selection predicate to perform queries with only inequality join conditions. In fact, this is the main reason why they cannot run for bigger datasets.

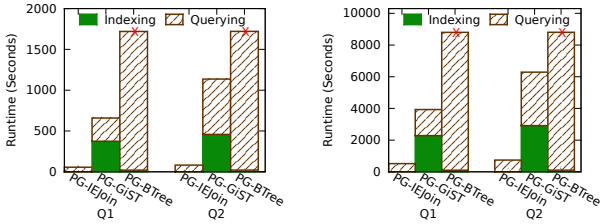
IEJoin **vs. indexing.** We now consider two different variants of PostgreSQL, each using a different indexing technique (GiST and BTree), to better evaluate the efficiency of our algorithm with bigger datasets in a centralized environment. We run Q_1 and Q_2 on datasets with 10M and 50M records. Figure 8 presents the results. In both experiments, IEJoin is more than one order of magnitude faster than PG-GiST. In fact, IEJoin is more than three times faster than the GiST indexing time alone. We stopped PG-BTree after 24 hours of runtime. Our algorithm performs better than these two baseline indices because it better utilizes the memory locality.

We observe that the memory consumption for MonetDB increases exponentially with the input size. For example, MonetDB uses 419GB for an input dataset with only 200K



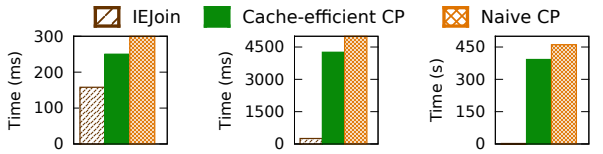
(a) Q_1 (b) Q_2

Figure 7: IEJoin **(centralized)**



(a) 10M rows (b) 50M rows

Figure 8: IEJoin **vs. BTree and GiST (centralized)**



(a) L1 cache (b) L2 cache (c) L3 cache

Figure 9: Q_1 **runtime for data that fits caches**

records. In contrast to MonetDB, IEJoin makes better use of the memory. Table 5 shows that IEJoin uses around 150MB for Q_1 (or less for Q_2) for an input dataset of 200K records (MonetDB requires two orders of magnitude more memory). Note that in Table 5, we report the overall memory used by sorted attribute arrays, permutation arrays, and the bit-array. Moreover, although IEJoin requires only 9.3GB of memory for an input dataset of 10M records, it runs to completion in less than one hour (3,128 seconds) for a dataset producing more than 7 billion output records.

We further analyze the breakdown time of IEJoin on the 50M rows datasets, as shown in Table 6. The table shows that, by excluding the time required to load the dataset into memory, scanning the bit-array takes only 40% of the overall execution time where the rest is mainly for sorting. This shows the high efficiency of our algorithm.

IEJoin **vs. cache-efficient Cartesian product.** We now push further our evaluation to better highlight the memory locality efficiency of IEJoin. We compare its performance with both naive and cache-efficient Cartesian product joins for Q_1 on datasets that fit the L1 cache (256 KB), L2 cache (1 MB), and L3 cache (8 MB) of the Intel Xeon processor. We used 10K rows for L1 cache, 40K rows for L2 cache, and 350K rows for L3 cache. Figure 9 reports the results of this experiment. When the dataset fits in the L1 cache, IEJoin is 1.8 times faster than the cache-efficient Cartesian product and 2.4 times faster than the naive Cartesian product. Furthermore, as we increase the dataset size of Q_1 to be stored at the L2 and L3 caches, we see that IEJoin becomes one and two orders of magnitude faster than the Cartesian product, respectively. This is because of the delays of L2 and L3 caches and the complexity of the Cartesian product.

Query	Input	Output	Time(secs)	Mem(GB)
Q_1	100K	9K	0.30	0.1
Q_1	200K	23K	0.28	**0.2**
Q_1	350K	68K	1.12	0.3
Q_1	10M	3M	67.5	7.6
Q_2	100K	0.2K	0.14	0.4
Q_2	200K	0.8K	0.28	**0.8**
Q_2	1M	42K	11.51	1.3
Q_2	**10M**	2M	92.53	**9.3**
Q_3	500k	154M	63.37	0.4
Q_3	1.5M	1B	453.79	1.1
Q_3	2M	2B	822.63	1.3
Q_3	4M	7B	**3,128.03**	3.0

Table 5: Runtime and memory usage (PG-IEJoin)

Query	Data reading	Data sorting	Bitarray scanning	Total time (secs)
Q_1	158	240	165	563
Q_2	319	332	215	866

Table 6: Time breakdown on 50M rows

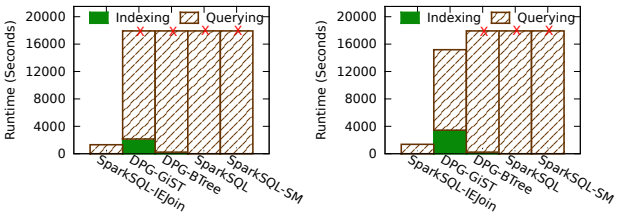
Single-node summary. IEJoin outperforms existing baselines by at least an order of magnitude for two main reasons: it avoids the use of the expensive Cartesian product and it nicely exploits memory locality by using memory-contiguous data structures with a small footprint. In other words, our algorithm avoids as much as possible going to memory to fully exploit the CPU speed.

7.4 Multi-node Experiments

We now evaluate our proposal in a distributed environment and using larger datasets.

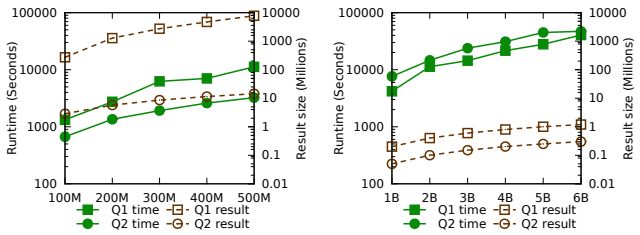
Distributed IEJoin **vs. baseline systems.** It is worth noting that we had to run these experiments on a cluster of 6 compute nodes only due to the limit imposed by the free version of the distributed PostgreSQL system. Additionally, in these experiments, we stopped the execution of any system that exceeds 24 hours. Figure 10 shows the results of all distributed systems we consider for queries Q_1 and Q_2. This figure again shows that our algorithm significantly outperforms all baseline systems. It is at least one order of magnitude faster that all other systems. In particular, we observe that only DPG-GiST could terminate before 24 hours for Q_2. In such a case, IEJoin is twice faster than the time required to run GiST indexing alone. These results show the high superiority of our algorithm over all baseline systems also in a distributed environment.

Scaling input size. We further push the evaluation of the efficiency in a distributed environment with bigger input datasets: from 100M to 500M records with large results size (Employees & Events), and from 1B to 6B records with smaller results size (Employees2 & Events2). As we now consider IEJoin only, we run this experiment on our entire 16 compute nodes cluster. Figure 11 shows the runtime results as well as the output sizes. We observe that IEJoin gracefully scales along with input dataset size in both scenarios. We also observe in Figure 11(a) that, when the output size is large, the runtime increases accordingly as it is dominated by the materialization of the results. In Figure 11(a), Q_1 is slower than Q_2 as its output is three orders of magnitude larger. When the output size is relatively small, both Q_1 and Q_2 scale well with increasing input



(a) Q_1 (b) Q_2

Figure 10: Distributed IEJoin (100M rows, 6 nodes)



(a) Employees & Events (b) Employees2 & Events2

Figure 11: Distributed IEJoin, 6B rows, 16 nodes

size (see Figure 11(b)). Below, we study in more details the impact of the output size on performance.

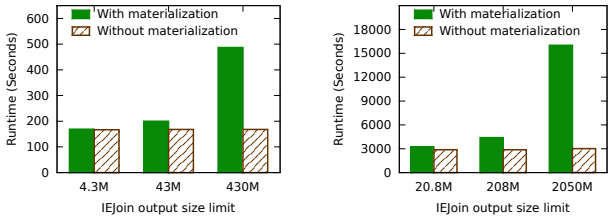
Scaling dataset output size. We test our system's scalability in terms of the output size using two real datasets (MDC and Cloud) as shown in Figure 12. To have a full control on this experiment, we explicitly limit the output size from 4.3M to 430M for MDC, and 20.8M to 2050M for Cloud. The figures clearly show that the output size affects the runtime; the larger the output size, the longer it will take to produce them. They also show that materializing a large number of results is costly. Take Figure 12(a) for example, when the output size is small (*i.e.,* 4.3M), materializing them or not will have similar performance. However, when the output size is big (*i.e.,* 430M), materializing the results takes almost 2/3 of the entire running time, as expected.

In order to run another set of experiments with much bigger output size, we created two variants of Q_3 for MDC data by keeping only two predicates over four (less selectivity). Figure 13 shows the scalability results of these experiments with no materialization of results. For Q_{3a}, IEJoin produced more than 1,000B records in less than 3,000 seconds. For Q_{3b}, we stopped the execution after 2 hours with more than 5,000B tuples in the temporary result. This demonstrates the good scalability of our solution.

Multi-node summary. Similarly to the centralized environment, IEJoin outperforms existing baselines by at least one order of magnitude. In particular, we observe that it gracefully scales in terms of input (up to 6B input tuples). This is because our algorithm first performs a join at the metadata level, which is orders of magnitude smaller than the actual data. As a result, it shuffles only those data partitions that can potentially produce join results. Typically, IEJoin processes a small number of data partitions.

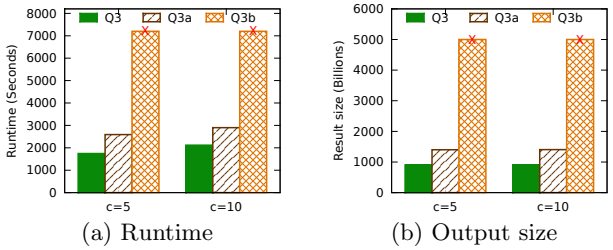
8. RELATED WORK

Several cases of inequality joins have been studied; these include band joins, interval joins and, more generally, spatial joins. IEJoin is specially optimized for joins with at least two predicates in {"<", ">", "≤", "≥"}.



(a) MDC - Q_3 (b) Cloud - Q_4

Figure 12: Runtime of IEJoin ($c = 10$)



(a) Runtime (b) Output size

Figure 13: Without result materialization ($c = 5, 10$)

A band join [?] of two relations R and S has a join predicate that requires the join attribute of S to be within some range of the join attribute of R. The join condition is expressed as $R.A - c_1 \leq S.B \ \& \ S.B \leq R.A + c_2$, where c_1 and c_2 are constants. The band-join algorithm [?] partitions the data from relations R and S into partitions R_i and S_i respectively, such that for every tuple $r \in R$, all tuples of S that join with r appear in S_i. It assumes that R_i fits into memory. Contrary to IEJoin, the band join is limited to a single inequality condition type, involving one single attribute from each column. IEJoin works for any inequality conditions and attributes from the two relations. While band join queries can be processed using our algorithm, not all IEJoin queries can run with a band join algorithm.

Interval joins are frequently used in temporal and spatial data. The work in [?] proposes the use of the relational Interval Tree to optimize joining interval data. Each interval intersection is represented by two inequality conditions, where the lower and upper times of any two tuples are compared to check for overlaps. This work optimizes non-equijoins on interval intersections, where they represent each interval as a multi-value attribute. Compared to our work, they only focus on improving interval intersection queries and cannot process general purpose inequality joins.

Spatial indexing is widely used in several applications with multidimensional datasets, such as Bitmap indices [?, ?], R-trees [?] and space filling curves [?]. In PostgreSQL, support for spatial indexing algorithms is provided through a single interface known as Generalized index Search Tree [?] (GiST). From this collection of indices, Bitmap index is the most suitable technique to optimize multiple attribute queries that can be represented as 2-dimensional data. Examples of 2-dimensional datasets are intervals (*e.g.,* start and end time in Q_2), GPS coordinates (*e.g.,* Q_3), and any two numerical attributes that represent a point in an XY plot (*e.g.,* salary and tax in Q_1). The main disadvantage of the Bitmap index is that it requires large memory footprint to store all unique values of the composite attributes [?, ?]. Bitmap index is a natural baseline for our algorithm, but, unlike IEJoin, it does not perform well with high cardinality attributes, as demonstrated in Figure 6. R-trees, on the

other hand, are not suitable because an inequality join corresponds to window queries that are unbounded from two sides, and consequently intersect with a large number of internal nodes of the R-tree, generating unnecessary disk accesses.

Several proposals have been made to speed-up join executions in MapReduce (e.g., [**?**]). However, they focus on joins with equalities and hence are forced to perform massive data shuffling to be able to compare each tuple with each other. There have been few attempts to devise efficient implementation of theta-join in MapReduce [**?**,**?**]. [**?**] focuses on pair-wise theta-join queries. It partitions the Cartesian product output space with rectangular regions of bounded sizes. Each partition is mapped to one reducer. The proposed partitioning guarantees correctness and workload balance among the reducers while minimizing the overall response time. [**?**] further extends [**?**] to solve multi-way theta-joins. It proposes an I/O and network cost-aware model for MapReduce jobs to estimate the minimum time execution costs for all possible decomposition plans for a given query, and selects the best plan given a limited number of computing units and a pool of possible jobs. We propose a new algorithm to do the actual inequality join based on sorting, permutation arrays, and bit arrays. The focus in these previous proposals is on efficiently partitioning the output space and on providing a cost model for selecting the best combination of MapReduce jobs to minimize response time. In both proposals, the join is performed with existing algorithms, which in the case of inequality conditions corresponds to Cartesian product followed by a selection.

9. ...THE END

To help Bob with his inequality join, we proposed two algorithms for the efficient evaluation of joins defined with inequality conditions. Our approach relies on auxiliary data structures that enable efficient computations and require small memory footprint. We presented a novel algorithm that exploits data locality in the data structures to achieve orders of magnitude speedup in the computation, and an optimized version of the same for self-joins. For both algorithms, we discussed extensions and optimizations. Finally, we presented both centralized and distributed versions of the algorithms, which are implemented on top of PostgreSQL and Spark SQL, respectively. Through extensive experiments over both synthetic and real data, we demonstrated that our solution is superior to baseline systems: it is 1.5 to 3 orders of magnitude faster than commercial and open-source centralized databases; and is at least 2 orders of magnitude faster than the original Spark SQL. More interestingly, we experimentally showed that, although theoretically the algorithm does not break the quadratic time bound, its performance is proportional to the size of the output. Future directions include the selectivity estimation for inequality join conditions to achieve better query optimization.

10. ACKNOWLEDGMENTS

Portions of the research in this paper used the MDC Database made available by Idiap Research Institute, Switzerland and owned by Nokia.